Database System Concepts

Chapter 15: Transactions

Departamento de Engenharia Informática
Instituto Superior Técnico

1st Semester
2009/2010

Slides (fortemente) baseados nos slides oficiais do livro "Database System Concepts"

Outline

Outline

Transaction Concept

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A transaction must see a consistent database
  - During transaction execution the database may be temporarily inconsistent
  - When the transaction completes successfully (is committed), the database must be consistent
  - After a transaction commits, the changes it has made to the database persist, even if there are system failures
  - Multiple transactions can execute in parallel
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

To preserve the integrity of data the database system must ensure:

- **Atomicity**: Either all operations of the transaction are properly reflected in the database or none are
- **Consistency**: Execution of a transaction in isolation preserves the consistency of the database
- **Isolation**: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished
- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Example

Transaction to transfer \$50 from account A to account B

1. read(A)
2. $A := A - 50$
3. write(A)
4. read(B)
5. $B := B + 50$
6. write(B)

Example (cont.)

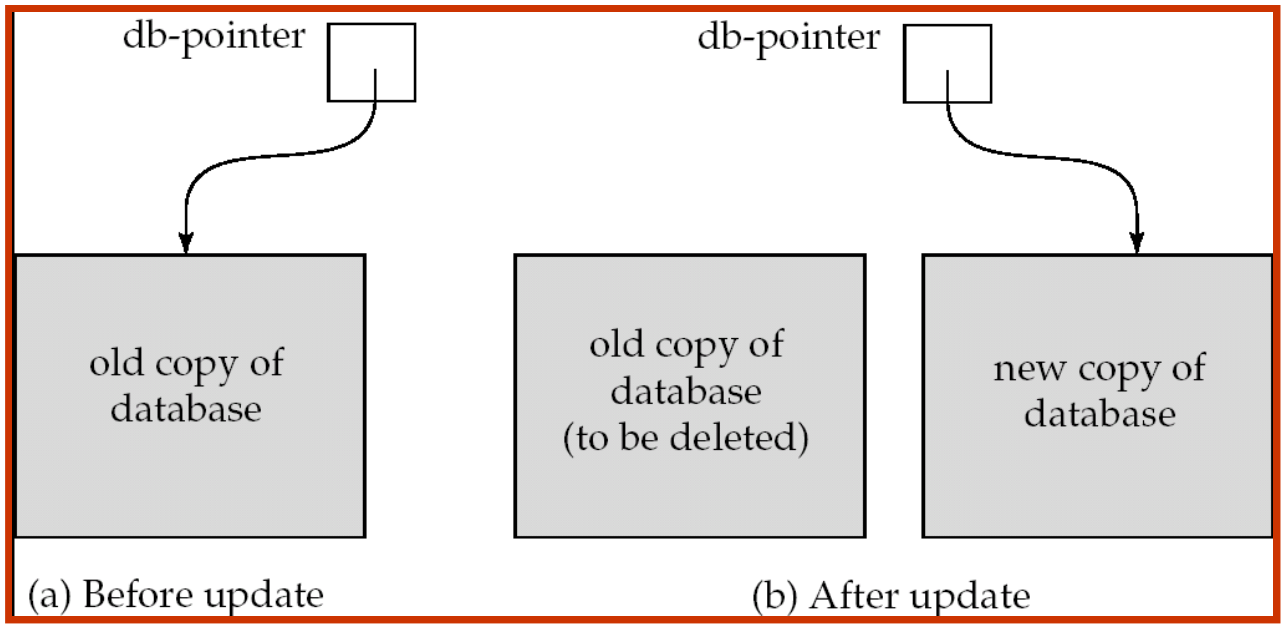
- Atomicity requirement - if the transaction fails between steps 3 and 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result
- Consistency requirement - the sum of A and B is unchanged by the execution of the transaction
- Isolation requirement - if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database
  - Isolation can be ensured trivially by running transactions serially, that is one after the other
  - However, executing multiple transactions concurrently has significant benefits
- Durability requirement - once the user has been notified that the transfer of the $50 has taken place, the updates to the database by the transaction must persist despite failures

Outline

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability
- Example: the shadow-database scheme:
  - assume that only one transaction is active at a time
  - a pointer called `db_pointer` always points to the current consistent copy of the database
  - all updates are made on a shadow copy of the database, and `db_pointer` is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk
  - in case transaction fails, old consistent copy pointed to by `db_pointer` can be used, and the shadow copy can be deleted

The Shadow-Database Scheme



- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases
  - does not handle concurrent transactions

Outline

Concurrent Execution

- Multiple transactions are allowed to run concurrently in the system
- Advantages:
  - increased processor and disk utilization, leading to better transaction throughput
    - one transaction can be using the CPU while another is reading from or writing to the disk
  - reduced average response time for transactions
    - short transactions need not wait behind long ones
- Concurrency control schemes - mechanisms to achieve isolation; that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- Schedule - a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a *commit* instruction as the last statement (will be omitted if it is obvious)
- A transaction that fails to successfully complete its execution will have an *abort* instruction as the last statement (will be omitted if it is obvious)

Schedule Example 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B

Serial schedule in which T_1 is followed by T_2

	T_1	T_2
1.	read(A)	
2.	$A := A - 50$	
3.	write(A)	
4.	read(B)	
5.	$B := B + 50$	
6.	write(B)	
7.		read(A)
8.		$t := A \times 0.1$
9.		$A := A - t$
10.		write(A)
11.		read(B)
12.		$B := B + t$
13.		write(B)

Schedule Example 2

A serial schedule where T_2 is followed by T_1

	T_1	T_2
1.		read(A)
2.		$t := A \times 0.1$
3.		$A := A - t$
4.		write(A)
5.		read(B)
6.		$B := B + t$
7.		write(B)
8.	read(A)	
9.	$A := A - 50$	
10.	write(A)	
11.	read(B)	
12.	$B := B + 50$	
13.	write(B)	

Schedule Example 3

A non-serial schedule equivalent to Schedule 1

	T_1	T_2
1.	read(A)	
2.	$A := A - 50$	
3.	write(A)	
4.		read(A)
5.		$t := A \times 0.1$
6.		$A := A - t$
7.		write(A)
8.	read(B)	
9.	$B := B + 50$	
10.	write(B)	
11.		read(B)
12.		$B := B + t$
13.		write(B)

Schedule Example 4

The following concurrent schedule does not preserve the value of (A + B)

	T_1	T_2
1.	read(A)	
2.	$A := A - 50$	
3.		read(A)
4.		$t := A \times 0.1$
5.		$A := A - t$
6.		write(A)
7.		read(B)
8.	write(A)	
9.	read(B)	
10.	$B := B + 50$	
11.	write(B)	
12.		$B := B + t$
13.		write(B)

Outline

1. Transactions
2. Atomicity and Durability
3. Concurrency
4. Serializability
   - Conflict Serializability
   - Testing Serializability
   - Recoverable Schedules
   - Concurrency Control
5. Transactions in SQL

Serializability

- **Basic Assumption:** each transaction preserves database consistency
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  - conflict serializability
  - view serializability

Note: we ignore operations other than *read* and *write* instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only *read* and *write* instructions.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q
  1. $I_i = \text{read}(Q), I_j = \text{read}(Q)$ — no conflict
  2. $I_i = \text{read}(Q), I_j = \text{write}(Q)$ — conflict.
  3. $I_i = \text{write}(Q), I_j = \text{read}(Q)$ — conflict
  4. $I_i = \text{write}(Q), I_j = \text{write}(Q)$ — conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule

	T_1	T_2
1.	read(A)	
2.	write(A)	
3.		read(A)
4.		write(A)
5.	read(B)	
6.	write(B)	
7.		read(B)
8.		write(B)

$\Leftrightarrow$

	T_1	T_2
1.	read(A)	
2.	write(A)	
3.	read(B)	
4.	write(B)	
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)

Conflict Serializability (cont.)

- Example of a schedule that is not conflict serializable:

	T_3	T_4
1.	read(Q)	
2.		write(Q)
3.	write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$

Other Notions of Serializability

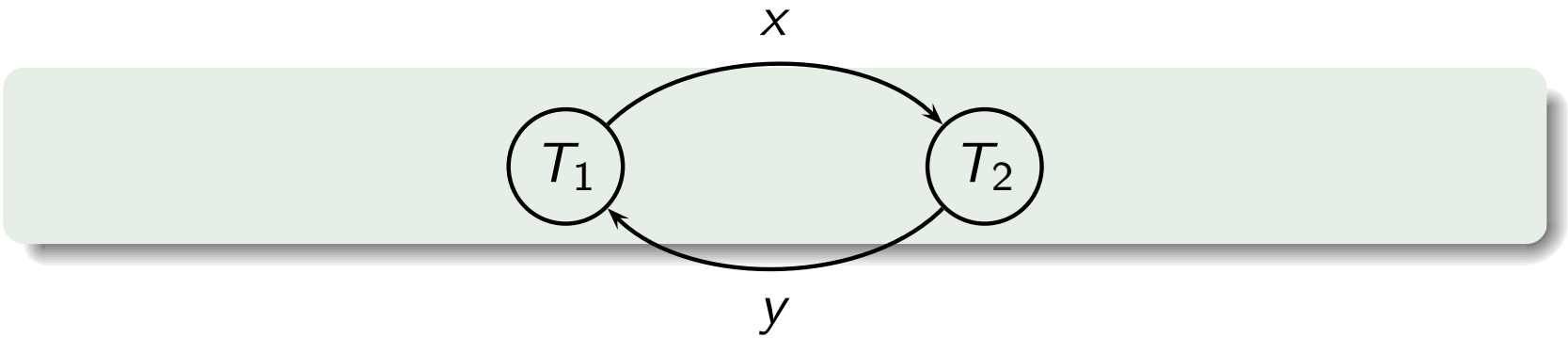
- The schedule below is serializable but not conflict-serializable

	T_1	T_5
1.	read(A)	
2.	$A := A - 50$	
3.	write(A)	
4.		read(B)
5.		$B := B - 10$
6.		write(B)
7.	read(B)	
8.	$B := B + 50$	
9.	write(B)	
10.		read(A)
11.		$A := A + 10$
12.		write(A)

- Determining such equivalence requires analysis of operations other than read and write

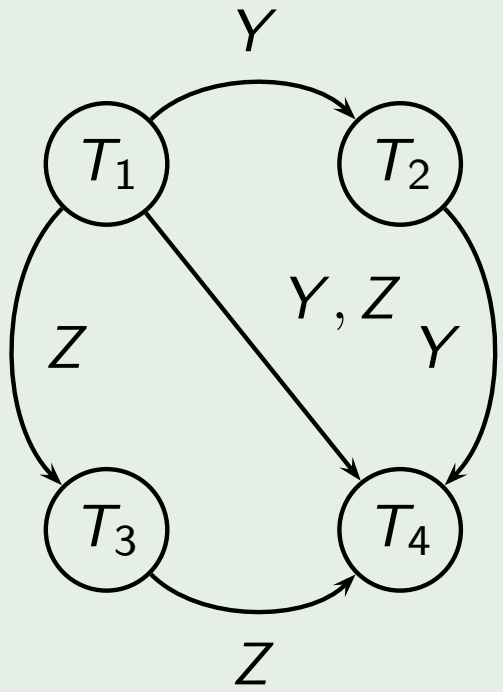
Testing for Serializability

- Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- Precedence graph - a directed graph where the vertices are the transactions
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier
- We may label the arc by the item that was accessed.



An Example

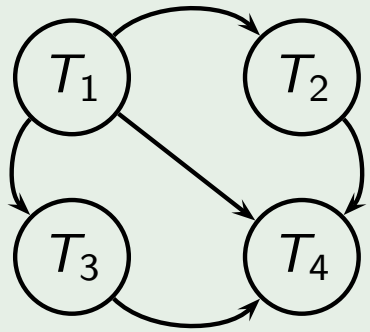
	T_1	T_2	T_3	T_4
1.		read(X)		
2.	read(Y)			
3.	read(Z)			
4.		read(Y)		
5.		write(Y)		
6.			write(Z)	
7.	read(U)			
8.				read(Y)
9.				write(Y)
10.				read(Z)
11.				write(Z)
12.	read(U)			
13.	write(U)			

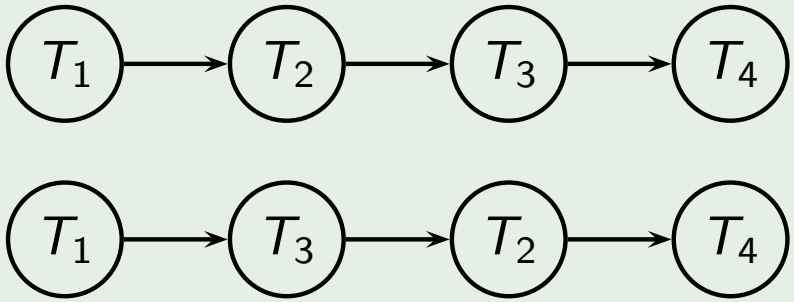


Testing for Conflict-Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic
- Cycle-detection algorithms exist which take order n^2 or $n + e$ time, where n is the number of vertices and e is the number of edges
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph
  - This is a linear order consistent with the partial order of the graph

Example





Recoverable Schedules

- Recoverable schedule - if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j
  - The following schedule is not recoverable

	T_8	T_9
1.	read(A)	
2.	write(A)	
3.		read(A)
4.		commit
5.	read(B)	
6.	...	

  - If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state
- The database must ensure that schedules are recoverable

Cascading Rollbacks

- Cascading rollback - a single transaction failure leads to a series of transaction rollbacks
  - Consider the following schedule where none of the transactions has yet committed

	T_{10}	T_{11}	T_{12}
1.	read(A)		
2.	read(B)		
3.	write(A)		
4.		read(A)	
5.		write(A)	
6.			read(A)

  - If T_{10} fails, T_{11} and T_{12} must also be rolled back
- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- Cascadeless schedules - cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict (or view serializable), and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability after it has executed is a little too late!
- **Goal:** to develop concurrency control protocols that will assure serializability

Concurrency Control and Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules
- Tests for serializability help us understand why a concurrency control protocol is correct
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur

Outline

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
  - **commit [work]** - commits current transaction and begins a new one
  - **rollback [work]** - causes current transaction to abort

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g. database statistics computed for query optimization can be approximate
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff of accuracy for performance

Levels of Consistency in SQL

- Serializable - default
- Repeatable read - only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable - it may find some records inserted by a transaction but not find others
- Read committed - only committed records can be read, but successive reads of record may return different (but committed) values
- Read uncommitted - even uncommitted records may be read

Lower degrees of consistency useful for gathering approximate information about the database

Levels of Consistency and Errors

Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**
Read uncommitted	✓	✓	✓
Read committed		✓	✓
Repeatable read			✓
Serializable			

- To set the consistency level in SQL:
  - **set transaction isolation level [serializable, ...]**

End of Chapter 15